IUT2 Université Grenoble Alpes
Département STID

Programmation Objet

R3.VCOD.10

Martin LENTSCHAT

martin.lentschat@univ-grenoble-alpes.fr

Notions avancées de POO

- Classe (avec ses Attributs et Méthodes) et Objet/Instance
- Envoi de messages
- Encapsulation
- **Héritage**
- Polymorphisme
- **Associations entre classes**
- **Agrégation** et **Composition**

IUT2 Université Grenoble Alpes
Département STID

Héritage multiple entre classes

- Certains langages de programmation permettent à une classe d'hériter de plusieurs classes : **héritage multiple**

- Exemple : un intervenant est à la fois un enseignant et un salarié du secteur privé
  - La classe *Intervenant* hérite à la fois de la classe *Enseignant* et de la classe *Salarié*

IUT2 Université Grenoble Alpes
Département STID

Héritage multiple entre classes

- Il faut redéclarer tous les attributs ou appeler les constructeurs des classes mères (pas de *super()*)

- En cas de méthodes ou d'attributs polymorphes, c'est l'ordre dans la déclaration de la classe qui est utilisé

  (ici : *Intervenant* > *Enseignant* > *Salarié*)

```python
class Enseignant:
    def __init__(self, matiere: str):
        self.specialite=matiere

    def test(self):
        print(__class__)


class Salarie:
    def __init__(self, money: int):
        self.salaire = money

    def test(self):
        print(__class__)


class Intervenant(Enseignant, Salarie):
    def __init__(self, matiere: str, money:  int, nom: str):
        # self.specialite = matiere
        Enseignant.__init__(self, matiere)
        #self.salaire = money
        Salarie.__init__(self, money)
        self.name = nom

    def to_string(self):
        return f"{self.name} enseigne {self.specialite} pour"\
               f"{str(self.salaire)} euros"


interv = Intervenant('HTML', 42, 'Tim Berners-Lee')
print(interv.to_string())
"""Tim Berners-Lee enseigne HTML pour42 euros"""
interv.test()
"""<class '__main__.Enseignant'>"""
```

IUT2 Université Grenoble Alpes
Département STID

Association - Agrégation

- Associe des objets en représentant une relation de *'Possède-Un'* (unidirectionnelle)
- Permet d'appeler les méthode de la classe associée
- Les objets existent indépendamment l'un de l'autre

```
class Room:
    def __init__(self, nom: str, porte):
        self.name = nom
        self.door = porte

    def get_name(self):
        return self.name

    def get_door(self):
        return self.door.get_dimensions()


class Door:
    def __init__(self, haut: float, large:  float):
        self.height = haut
        self.width = large

    def get_dimensions(self):
        return self.height, self.width


d = Door(1.95, 0.85)
b107=Room('B107', d)
print('La porte de la pièce ', b107.get_name(), ' fait ',
      b107.get_door(), 'mètres')
"""La porte de la pièce  B107  fait  (1.95, 0.85) mètres"""
```

IUT2 Université Grenoble Alpes
Département STID

Association - Composition

- Associe des objets en représentant une relation *'Partie-de'* (unidirectionnelle)

- Permet d'appeler les méthode de la classe associée

- Le composant (contenu) ne peut exister sans le composé (conteneur).

```python
class Room:
    def __init__(self, nom: str, dims):
        self.name = nom
        self.door = Door(dims[0], dims[1])

    def get_name(self):
        return self.name

    def get_door(self):
        return self.door.get_dimensions()


class Door:
    def __init__(self, haut: float, large:  float):
        self.height = haut
        self.width = large

    def get_dimensions(self):
        return self.height, self.width


# composition
b107=Room('B107', [1.95, 0.85])
print('La porte de la pièce ', b107.get_name(), ' fait ',
      b107.get_door(), 'mètres')
"""La porte de la pièce  B107  fait  (1.95, 0.85) mètres"""
```

IUT2 Université Grenoble Alpes
Département STID

Classes abstraites

- Permises via le module `ABC`
- Permet de différencier des méthodes :
  - abstraites, devant être implémentées dans toutes les sous-classes
  - concrètes, partagées entre les sous-classes
- Une classe abstraite ne peut pas être instanciée

```python
from abc import ABC, abstractclassmethod

class Pokemon(ABC):
    @abstractclassmethod
    def get_attack(self):
        pass

    def talk(self):
        return f'{self.name}!'.capitalize()*2


class Pikachu(Pokemon):

    def __init__(self, nom: str, dmg: int):
        self.name = nom
        self.attack = dmg

    def get_attack(self):
        return self.attack

# p = Pokemon()
# TypeError: Can't instantiate abstract class Pokemon
#            with abstract methods get_attack

p = Pikachu('pika', 18)
print(p.get_attack())
"""18"""
print(p.talk())
"""Pika!Pika!"""
```

IUT2 Université Grenoble Alpes
Département STID

Décorateurs

- Les décorateurs sont utilisés afin de modifier le comportement d'une fonction, méthode ou classe (e.g. *@abstractclassmethod*)
- Exemples:
  - *@property* → spécifie un *getter*
  - *@**x**.setter* → spécifie un *setter* de l'attribut ***x***
  - *@classmethod* → en fait une méthode de classe
  - *@staticmethod* → en fait une méthode statique (liée à la classe mais sans accès aux attributs)
  - @typing.final (`typing`) → ne peut pas être hérité

- Mon preféré :
  - *@lru_cache* (`functools`) → met les résultats en cache dans un dictionnaire